MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1322 October 1991

A Comparative Analysis of Reinforcement Learning Methods

Maja J Mataric

Abstract

This paper analyzes the suitability of reinforcement learning for both programming and adapting situated agents. In the the first part of the paper we discuss two specific reinforcement learning algorithms: Q-learning and the Bucket Brigade. We introduce a special case of the Bucket Brigade, and analyze and compare its performance to Q-learning in a number of experiments. The second part of the paper discusses the key problems of reinforcement learning: time and space complexity, input generalization, sensitivity to parameter values, and selection of the reinforcement function. We address the tradeoff between the amount of built in and learned knowledge in the context of the number of training examples required by a learning algorithm. Finally, we suggest directions for future research.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part by the Mazda Corporation, in part by the University Research Initiative under Office of Naval Research contract N00014–86–K–0685, and in part by the Advanced Research Projects Agency under Office of Naval Research contract N00014–85–K–0124.

1 Introduction

Reinforcement learning (RL) has recently grown in popularity as the learning methodology of choice in the situated agent community. RL is appealing because it allows the agent to adapt to its environment as it gains information over time. It is particularly well suited for action learning, which is the main concern in control of situated agents.

However, reinforcement learning suffers from a number of problems which are in conflict with the goals of situated agent control. This paper analyzes the suitability of the general approach by using an in depth comparison of two RL methods: Q-learning, introduced by Watkins [**Watkins 89**] and the classifier system (CS) Bucket Brigade, introduced by Holland [**Holland 85**]. The first part of the paper introduces and compares the two methods. The second part of the paper discuss the properties of reinforcement learning, as demonstrated by the example algorithms, their weaknesses, and their role within the space of various learning approaches.

2 A Definition of Reinforcement Learning

Reinforcement learning (RL) addresses the problem of learning a mapping (also called a policy or action map) between all of the states the system can be in and all of the actions it can execute using the reward and punishment received from the world. The goal is for the agent to learn to select the right action in each state. The inputs to the learner are the state of the world and the reinforcement signal, and its outputs are actions. Perfect state information is assumed. The problem is to find a function which closely enough approximates the mapping between all of the states the agent can perceive, and all of the actions it can take. The method is some form of search of the space of possible functions. RL algorithms have been used to learn tasks ranging from pushing a box without getting stuck to balanced walking and navigating through a maze.

The following is the general form of an RL algorithm [**Kaelbling 90**]:

1. Initialize the learner's internal state I to I_0.
2. Do Forever:
   a. Observe the current world state s.
   b. Choose an action $a = F(I, s)$ using the evaluation function F.
   c. Execute action a.
   d. Let r be the immediate reward for executing a in world state s.
   e. Update the internal state $I = U(I, s, a, r)$ using the update function U.

The internal state, I, encodes the information the learning algorithm saves about the world, usually in the form of a table maintaining state and action data. The update function U adjusts the current state based on the received reinforcement, and maps the current internal state, input, action, and reinforcement into a new internal state. The evaluation function F maps an internal state and an input into an action based on the information stored in the internal state. The different RL algorithms vary in their definitions of U and F.

The above framework assumes that, at each time step, the agent receives immediate reinforcement, the *complete* information about the value of the last action it took. In the general case, reinforcement can be arbitrarily delayed, and the problem of assigning reward or punishment to a state based on delayed reinforcement is termed *temporal credit assignment*. The first statement of the problem is due to [**Samuel 59**], whose checkers-learning program dealt with deciding which moves to reward for eventually leading to "a triple jump."

Temporal credit can be assigned in two ways: either the reward is appropriated to all of the state-action pairs *after* it is received, or an expected value of the *future* reward is calculated and maintained incrementally. The latter approach leads to a class of delayed reinforcement algorithms termed *temporal difference* (TD) methods which assign credit locally based on the difference between temporally successive predictions [**Sutton 88**]. Both Q-learning and the Bucket Brigade are instances of TD.

While temporal credit assignment deals with propagating reward backward in time, *structural credit assignment* deals with propagating the reward across similar states in order to couple them with similar actions. All RL approaches rely on exploring the complete state space, which is exponential in the size of the input vector. Consequently, input generalization, the ability to collapse similar input states together, is critical in making the approach computationally feasible.

Specific methods for dealing with both temporal and structural credit assignment will be described and analyzed in subsequent sections.

3 Q-learning

Q-learning is a reinforcement learning algorithm based on delayed reinforcement [**Watkins 89**]. The goal of the algorithm is to, at each time step, maximize $Q(s, a)$, the expected discounted reward of taking action a in the input state s. The algorithm maintains and updates a table of Q values, one for each state-action combination. The utility E of any state is the maximum Q value of all actions that can be taken in that state. The Q value of doing an action in a state is defined as the sum of the immediate reward r and the utility $E(s')$ of the next state s' according to the state transition function T, discounted by the parameter γ.

Formally:

$$s' \leftarrow T(s, a)$$
$$E(s) = \max_a Q(s, a)$$
$$Q(s, a) = r + \gamma E(s'),\ 0 \leq \gamma \leq 1$$

Q values are updated by the following rule:

$$Q(s, a) \leftarrow Q(s, a) + \beta(r + \gamma E(s') - Q(s, a))$$
$$0 \leq \beta \leq 1$$

An RL algorithm using Q-learning has the following form:

1. Initialize all $Q(s,a)$; select s_0.
2. Do Forever:
   a. Observe the current world state s.
   b. Choose an action a that maximizes $Q(s,a)$.
   c. Execute action a.
   d. Let r be the immediate reward for executing a in state s.
   e. Update $Q(s,a)$ according to the rule above. Let the new state be $s' \leftarrow T(s,a)$.

The key drawbacks of Q-learning are its sensitivity to the parameter values and the reinforcement function, and its time and space complexity mandated by the state space and the Q table which must be maintained.

The choice of β and γ, the key parameters in Q-learning, affects the efficiency of the learner. β determines the learning rate; $\beta = 1$ results in an update rule which disregards all history accumulated in the current Q value. It resets Q to the current sum of the received and expected reward at every time step, which usually causes the algorithm to oscillate.

γ is the discount factor for future reward. Ideally, γ should be as close to 1 as possible so that the relevance of future reward is maximized. In a deterministic world, γ can be set to 1, but in the general case two algorithms with $\gamma = 1$ cannot be compared since, in the limit, the expected future reinforcement of both will go to ∞.

The initial Q values can affect the speed of convergence. Intuitively, if the table is initialized close to the optimal policy, this will speed up the learning process. Of course, the optimal Q values are not known *a priori*. If initialized to 0 in a problem whose optimal policy has positive final Q values, the algorithm will converge to the first positive value, never exploring other possibilities [**Kaelbling 90**]. This can be remedied by occasionally performing a random action to guarantee that the entire action space is eventually explored. A better solution is to initialize the Q values to be higher than their anticipated optimal values and gradually decrease them.

Q-learning is sensitive to the coupling between the initial Q values and the reinforcement function. If the function is positive and the table is initialized to values exceeding the optimal policy, the system will take longer to converge than if the reinforcement function contains some negative signals.

Finally, the convergence of Q-learning requires a large number of trials, i.e. the algorithm relies on an infinite number of visits to the same state (for the proof see [**Watkins 89**]). This is a key drawback of classical Q-learning: it takes too long to converge for any non-trivially sized input vector.

4 Reinforcement Learning in Classifier Systems

We now turn to another instance of RL which, on the surface, appears rather different from Q-learning but shares some critical similarities. A *classifier system* (CS) is an adaptive production rule system consisting of a fixed number of condition-action pairs called classifiers [**Holland 86**]. The conditions are encoded as fixed-length bit strings over the alphabet $\{0, 1, \#\}$, where # is the default or "don't care" symbol. The action of a classifier consists of posting its message to a global board, which may result in an action to be performed in the world. At each time step, the messages on the board are matched to the conditions of all classifiers in parallel, and all satisfied classifiers make bids to post their messages to the board next. The highest bidders win and their messages are posted.

Classifier systems perform two types of learning: what classifiers to have (classifier generation) and what classifiers to activate (classifier reinforcement). *Genetic algorithms* are a class of methods for classifier generation. They employ mutation and crossover on the classifier population in order to, over time, evolve increasingly more "fit" classifiers. Widely discussed in the literature (e.g. [**Goldberg 89**]Goldberg85), genetic algorithms will not be addressed here. Instead, we will concentrate on classifier reinforcement, the process of assigning strengths to classifiers based on the reward they receive over time.

4.1 The Bucket Brigade Algorithm

The *Bucket Brigade* is a temporal differencing reinforcement learning algorithm for propagating reward down a chain of classifiers. Whenever reward is received, it is divided among the classifiers whose firing enabled it. Since reward is not received at every time step, the strength of a classifier is adjusted based on its "distance" from the reward. The closer the classifier in the chain to the reward, the more strength it receives. The classifiers whose firing was immediately followed by reinforcement divide the reward. Next, the classifiers that enabled them receive a smaller share of the reward, and so on down the chain.

Initially, all classifiers are assigned equal strength S. When a classifier C matches a message on the board, it posts a bid B proportional to its strength and its specificity. The specificity H of a classifier is the ratio between the number of specified (non-#) bits and the total number of bits in the classifier's condition. If C wins the bidding, it gets to post a message to the board next, and its strength is decreased by the magnitude of its bid. If its message causes an external action, its strength is increased by a portion of the received reinforcement r.

Besides by immediate reinforcement, a classifier's strength is increased by the bids of its successors. If a classifier C posts a message which is, in the next time step, matched by another classifier C', and C' then wins a bid, the strength of C is increased by the amount of C''s bid. If multiple classifiers contributed to C''s match, they split the bid evenly.

Formally:

$M(C)$ = number of messages matched by C
n = condition length in bits
$H(C)$ = (number of non-#'s)$/n$
$B(C,t) = cH(C)S(C,t)$, where $0 \leq c \leq 1$

Classifier strength is updated by the following rule:

$$S(C,t+1) \leftarrow S(C,t) + r - B(C,t) + B(C',t+1)/M(C')$$

An RL algorithm using the Bucket Brigade has the following form:

1. Initialize all S(C); select some C and post its message m on the board.
2. Do Forever:
   For each message m on the board:
   a. Match m to all classifiers.
   b. Compute $B(C)$ for all C that match m; select the winners.
   c. Post the winners' messages m_{new} on the board.
   d. Let r be the immediate reward for posting m_{new}.
   e. Update $S(C)$ according to the rule above.

4.2 Long Classifier Chains

Typical for production rule systems, the number of rule firings (or classifier activation) required to connect an input and an output of a classifier system can be unbounded [**Kaelbling 90**]. The longer the classifier sequence, the longer it takes to update the strengths of the early classifiers, i.e. the more times the system must go through the same classifier sequence. This makes Bucket Brigade systems slow to adapt to changes in the environment.

In order to speed up learning, [**Holland 85**] suggests the use of a "bridging" or "epoch marking" classifier that is activated by the first classifier in a sequence, and remains active until the end of the sequence when external reinforcement is received. At this time, the epoch marker receives a large amount of reinforcement. When the chain is activated again, the epoch marker passes some of its strength directly to the first classifier in the sequence. Since its strength is high, the fraction it passes on to the front of the chain significantly upgrades the strength of the first classifier. This speeds up reward propagation from a long chain to a single step.

Classifier sequences can be divided into two main types: *reflex* and *non-reflex*. Reflex sequences are simple chains in which each of the classifiers is activated solely by the message of its predecessor. Non-reflex sequences contain classifiers which are activated by more than one predecessor, i.e. more than one condition matches the posted message. [**Riolo 87**] shows that strengths of non-reflex classifiers fall off exponentially with the length of the chain. He gives experimental evidence that the use of bridging classifiers greatly expedites strength learning in both reflex and non-reflex sequences.

4.3 Default Hierarchies

The classifier system solution to decreasing the size of the state space is by categorizing states into abstraction hierarchies. The categorization emerges from the presence of #, the "don't care" symbol in the classifier condition alphabet which allows for different levels of rule specificity in the system. The more #'s a classifier contains, the more general it is (e.g. (1 0 #) is more general than (1 0 1)), and the more conditions it will match. The rules which match the same conditions and are more specific, are termed "exceptions" and belong to a subset of conditions matched by a more general parent. Over time, as more rules are added to the system, a hierarchy emerges in which increasingly more specific rules serve as exceptions to the more general classifiers. Through the bidding system, the more general rules are less likely to be correct, causing the formation of more specific rules. The system of default hierarchies allows a CS to learn incrementally.

An alternative to emergent hierarchies is to categorize the states by hand. [**Wilson 87**] suggests such as approach. Although no general method for constructing a hierarchy is given, the more domain specific knowledge is employed the more useful the hierarchy structure can be made.

5 Q-learning vs. the Bucket Brigade

Reinforcement learning is a form of gradient descent (or hill climbing) in parameter space. Specifically, the goal of RL algorithms is to minimize the parameter error, i.e. to maximize received reinforcement over time. Both Q-learning and the Bucket Brigade are gradient descent strategies. They both perform temporal and structural credit assignment by keeping track of combinations of states and actions. Both perform search in the space of "strength" functions mapping states to actions. In order to compare their performance, we next introduce a special case of the standard Bucket Brigade algorithm.

Classifier systems couple two types of learning: reinforcement learning and genetic learning. The reinforcement learning (Bucket Brigade) orders the classifiers by strength S. The genetic learning discards the weakest of the classifiers, and generates new ones by applying mutation and crossover operations on the strongest. In order to compare Bucket Brigade to Q-learning, the reinforcement portion of the CS needs to be isolated, so the genetic portion is removed. However, the purpose of the reinforcement part of CS is to provide strengths for the genetic learner. Since classifiers now cannot be added and removed from the system, they must all be supplied initially. The system is initialized with a set of classifiers $C(s,a)$ such that s is the condition or the n-bit input state, and a is the action. Consequently, there is a total of $2^n|a|$ classifiers, all of which are fully specified. Thus, #'s are eliminated, and P, the measure of specificity of a classifier, is the same for all C's, so the P term is dropped from the bid equation.

Q-learning and the Bucket Brigade both deal with the problem of propagating reward down a chain of states. The key difference is that Q-learning uses the maximum discounted future reward, whereas the Bucket Brigade computes the current reward, and then propagates it to the previous state. The following formalism allows for implementing Q's maximization within the Bucket Brigade:

$C = (s,a)$ where s is a state and a is an action.
$|C| = 2^n|a|$ and $\forall C[P(C) = 1]$
$S(s,a,t+1) = S(s,a,t) + r - B(s,a,t)$
$S(C) = S(C) + B(C')$ where $C' \leftarrow T(C)$

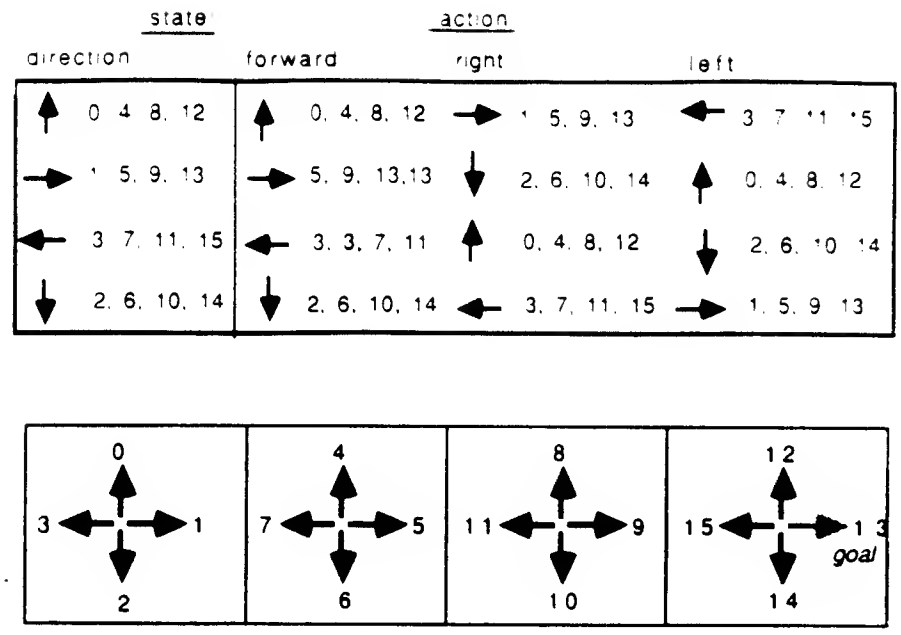

Figure 1: The learning task consists of 16 states, one of which is the goal. In each state three actions are possible: move forward, turn left, and turn right. Attempting an action against a boundary does not change the state. The shown state transition function implements simple motion so the task can be visualized as two-dimensional navigation.

$$B(s,a,t) = \begin{cases} cS(C,t) & \text{if } C(t) = \max_a S(s,a,t) \\ 0 & \text{otherwise} \end{cases}$$

At each time step, the current classifier $C'(s,a)$ receives the immediate reinforcement, and pays the bid proportional to the strength of the best action to be taken from that state, i.e. the maximum strength classifier that matches the current state s. This is the key change: instead of paying a bid proportional to its own strength $S(s,a,t)$, C' pays proportional to the strongest of the classifiers $\max_a S(s,a,t)$ that match the current state. In the same time step, the predecessor C, whose message was matched by the current classifier C', receives the value of C''s bid.

This special case of the Bucket Brigade (SBB) implements Q-learning in two steps. While Q-learning updates the Q value of the current state based on the maximum of the next state, SBB updates the previous state with the maximum of the current state.

SBB implements reflex sequences. Instead of bidding, the next action is selected so as to maximize the received reward (i.e. the one which uses a classifier with the most strength). Only one classifier is active at a time, unlike standard CS in which multiple classifier can compete in parallel. Furthermore, the current state receives the reinforcement and passes it back to the previous state, so the bid is not shared but goes straight to the predecessor in the action chain.

5.1 An Example

The following learning problem was used for comparing Q-learning and SBB. The world consists of 16 states, one of which is the goal, and three possible actions (going forward, turning left by 90 degrees, and turning right by 90 degrees), all of which can be tried in all states. The state transition function is defined so that the problem can be visualized as two-dimensional navigation in a row of four tiles, each of which contains four perceptual states. Attempting an action against a boundary does not change the state. Figure 1 illustrates the task and the state transition function.

Action Selection: In both algorithms, actions were selected so as to maximize the Q or S value. In the goal state, a random action was selected in order to force the learner to escape the potential well which would keep it stuck at the goal where both Q (or S) and the received reinforcement are maximized. In order to converge to the optimal policy, the agent must explore the entire state space, rather than stay at the goal once it reaches it. It is not enough to select a random action with some small probability τ. Unless τ is relatively large, the accumulated probability of the agent escaping the potential well is too small to allow for learning the policy in a reasonable number of trials.

Table Update: In order to propagate the strength values for each state-action combination, the entries in the table must be updated in one of two ways. Either a state is updated as it is visited by the agent (this is the implementation we chose), or the changes are propagated through the table for a chosen number of states at each time step. For example, [**Mahadevan and Connell 90**] uses a five-step update process. The later solution speeds up the learning by a constant factor. Even if all of the states in the table are updated at each time step, the agent can still get stuck at the goal, illustrating that the update function is not related to the potential well problem.

State Transition: The following state transition function was used:

$$p(T(x \rightarrow y)) = \begin{cases} 0.9 & \text{if } T(x) = y \\ 0.1 & \text{otherwise} \end{cases}$$

A random state transition was selected 10% of the time.

The algorithms were tested on the same problem, the same optimal policy, two different parameter values, and three different reinforcement functions. The data plots show individual runs of the learning algorithms as crosses. The y-axis in each plot indicates the number of time steps to convergence to the optimal policy, while the x-axis shows the different values of the parameters being tested. For each set of runs of an algorithm with particular parameter settings, the mean number of time steps to convergence is indicated with a bullet, and the standard deviation is shown with a vertical line. The algorithms showed sensitivity to the randomness inherent in both of the learning rules. This sensitivity was manifested by large standard deviations in almost all experiments.

Q-learning Performance: Figure 2 illustrates the performances of Q-learning using three different initial values for the Q table, while figure 3 shows its performance on two different initial states. The performance of the algorithm is not significantly affected by either of the parameters. Although the measured performance varies in both the mean and the standard deviation, the variation is not significant compared to the average variance

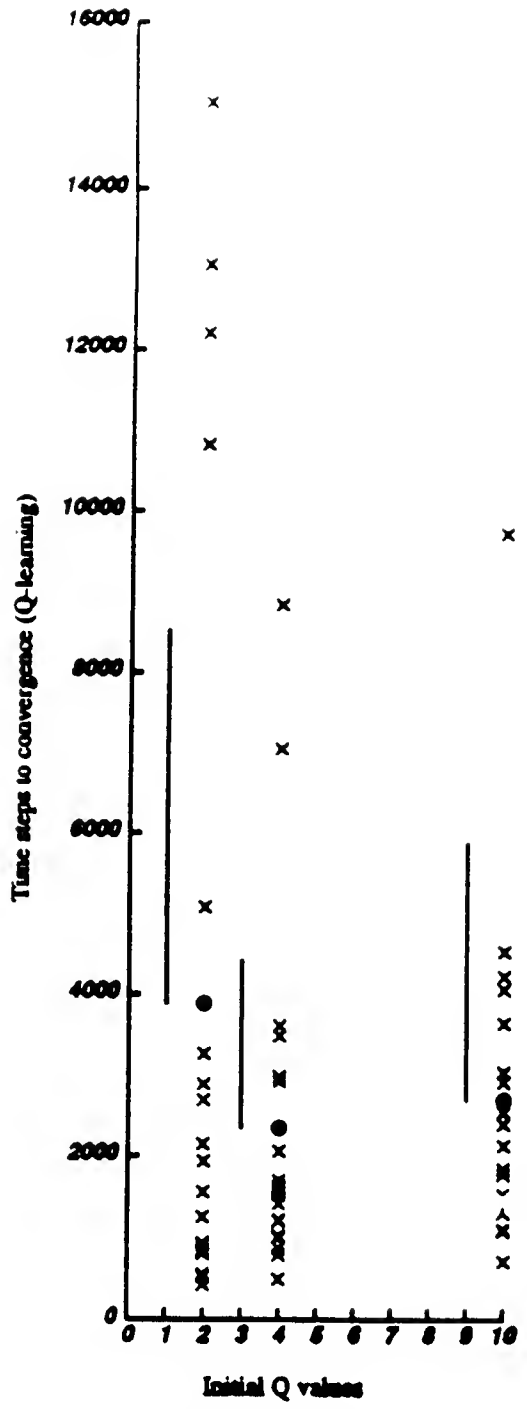

Figure 2: The graph illustrates the performance of Q-learning given three different initial values for the Q table. The smaller of the values was in range of the optimal policy, while the other exceeded it.

between individual trials over a large number of runs. Figure 4 illustrates the algorithm's apparent insensitivity to small changes in the learning rate parameter β. As shown in figure 5, Q-learning is very sensitive to the value of γ, the future reward discount factor. This parameter determines how much influence future reward will have on the current state. In deterministic worlds, such as the one used here, it is useful to set γ close to 1 in order maximize the value of future information at each time step. Consequently, the higher value of γ speeds up the learning.

SBB Performance: Figure 6 illustrates the performance variation for two different initial S table values. The smaller of the values was in the range of the optimal policy, so a few of the initial S values were equal to their target values. Not surprisingly, this resulted in somewhat faster mean convergence time and a significantly smaller standard deviation. However, in general it is not possible to have a good *a priori* estimate of the optimal policy, thus making the problem of initializing the S values (and Q values) difficult. Figure 7 shows the algorithm's performance starting from three different initial states, one of which was the goal state. The plot shows no significant dependence on this parameter. Figure 8 illustrates the algorithm's sensitivity to the value of c, the fraction of the received strength that is propagated to the previous classifier. The larger value of c increases the mean convergence time by over an order of magnitude. The higher the value of c, the more weight is given to each trial, causing the algorithm to rely on the "local" decision and oscillate around the optimal policy before being able to converge on it.

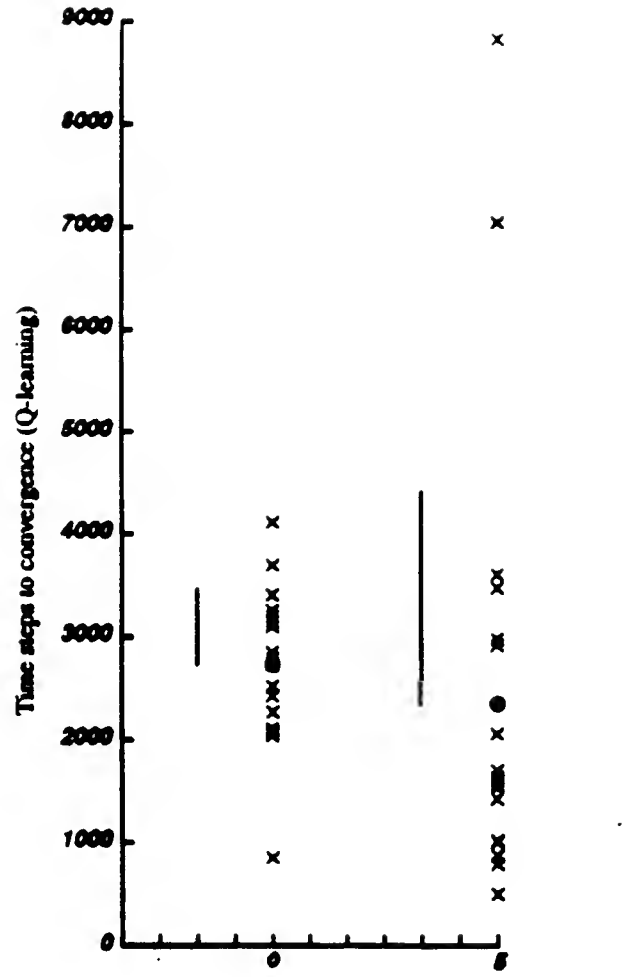

Figure 3: The graph shows the performance of Q-learning started in two different initial states.

Comparison: Both algorithms were insensitive to initial Q and S values and start states, and sensitive to γ and c, the parameters weighting the value of each time step. γ and c can be viewed as duals of each other. A high value of γ puts more importance on future trials. Similarly, a low value of c decreases the weight of the immediate S values, effectively increasing the importance of future reward. The relative convergence times for Q-learning and Bucket Brigade were comparable for analogous scaling of those two parameters.

In the shown trials, both algorithms were tested on a three-valued reinforcement function with small variance ($r \in \{-1, 0, 5\}$), shown on the top of figure 9. When tested on an impulse function ($r \in \{0, 3000\}$) shown on the bottom of figure 9, the performance of both algorithms declined by several orders of magnitude. However, our experiments demonstrated that the standard classifier system configuration of the Bucket Brigade algorithm favors the impulse reinforcement function.

Finally, figure 10 comparses the performance of Q-learning and the SBB algorithm using the same state

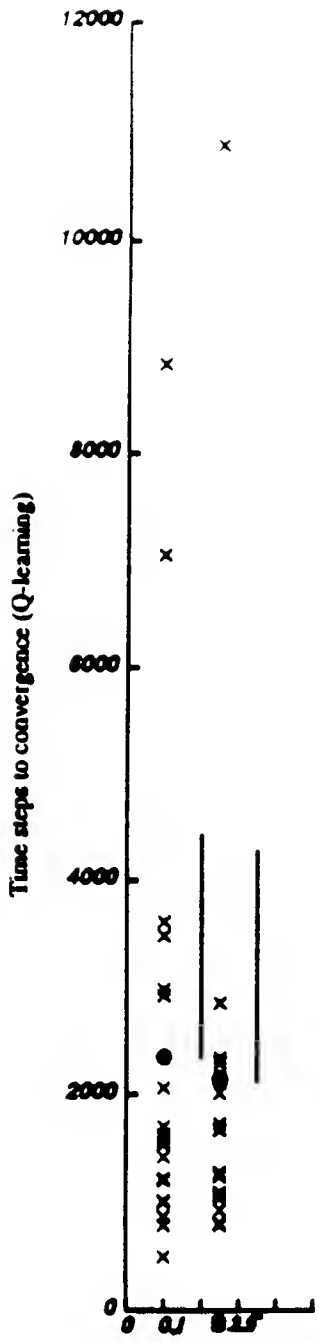

Figure 4: The graph plots the performance of Q-learning with two different values of β, the learning rate.

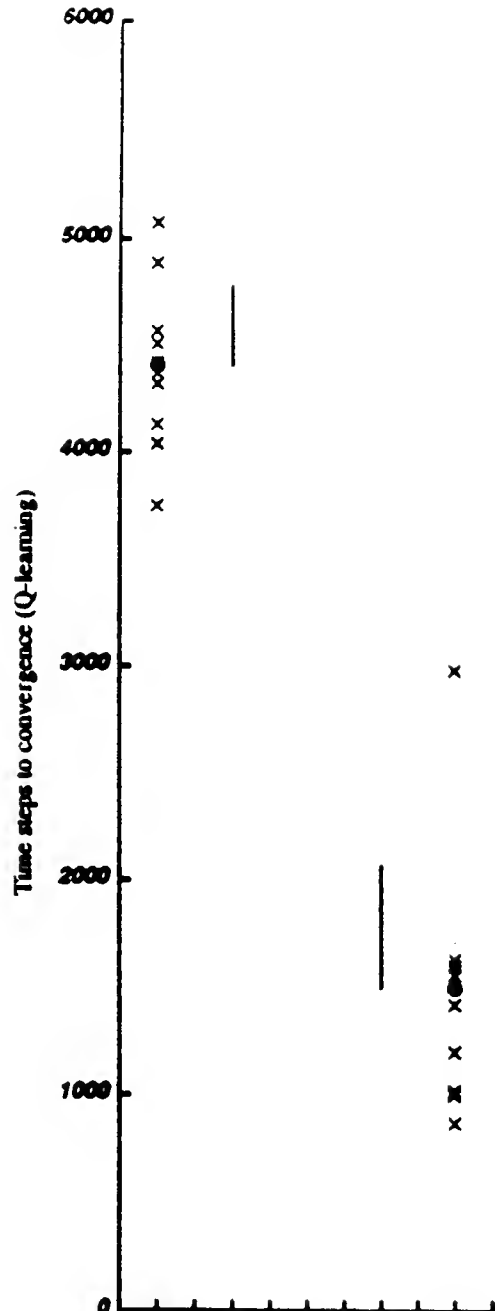

Figure 5: The graph illustrates the performance of Q-learning on two different values of γ, the expected future reward discount factor.

space, the same initial state and initial values for the Q and S tables, dual values of γ and c ($\gamma = 0.9$, $c = 0.1$), and the same reinforcement function (3-valued). On the shown problem, the special case of the Bucket Brigade outperforms Q-learning by approximately an order of magnitude in the number of time steps required for convergence to the optimal policy. However, further experimentation showed that the use of even a slightly modified reinforcement function (e.g. scaling the function shown in figure 9 by one to ($r \in \{0, 1, 6\}$)) reversed the performance results.

The two algorithms need to be tested on a much larger number of trials and on different learning problems before conclusions can be made about their performance differences. Further, a characterization of the parameter interaction is needed for proper analysis. But observation of the data alone illustrates the algorithms' similar, and similarly uncharacterized, sensitivity to the learning parameters and to the unavoidable randomness inherent in the approach.

6 Input Generalization

Our simple example of the two RL algorithms demonstrates that, even in a 16-state world, the number of trials to convergence is prohibitively large. Indeed, the exponential relationship between the size of the input vector and the size of the state space is a key problem in reinforcement learning. It introduces both temporal and spatial constraints on the size of the learning problems that can be addressed. Specifically, Q-learning is a table-based scheme, which necessitates keeping statistics about all of the states, which results in a tremendous memory requirement. Additionally, the larger the state space, the slower the system will be in converging to the desired policy. Q-learning requires visiting all of the states infinitely many times which, for most realistic problems, takes too long, even in simulation, and unrealistically long if the experiments are performed in the physical world. Finally, the larger the ratio between the number of states and the reinforcement, the slower the

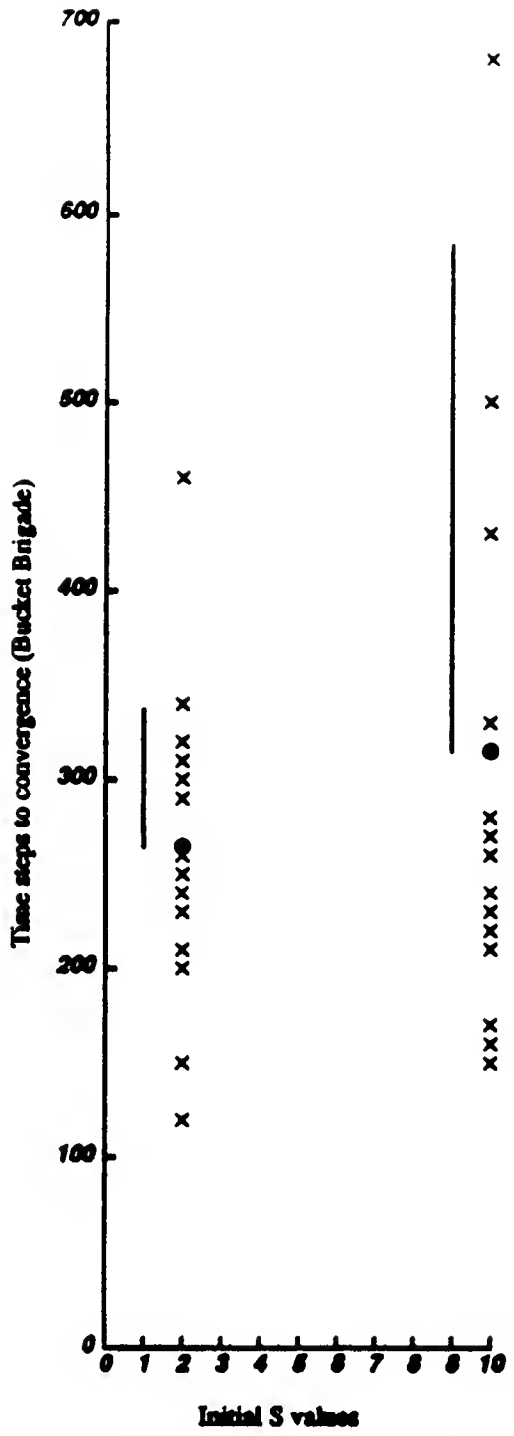

Figure 6: The graph illustrates the performance of the Bucket Brigade with different initial values in the S table.

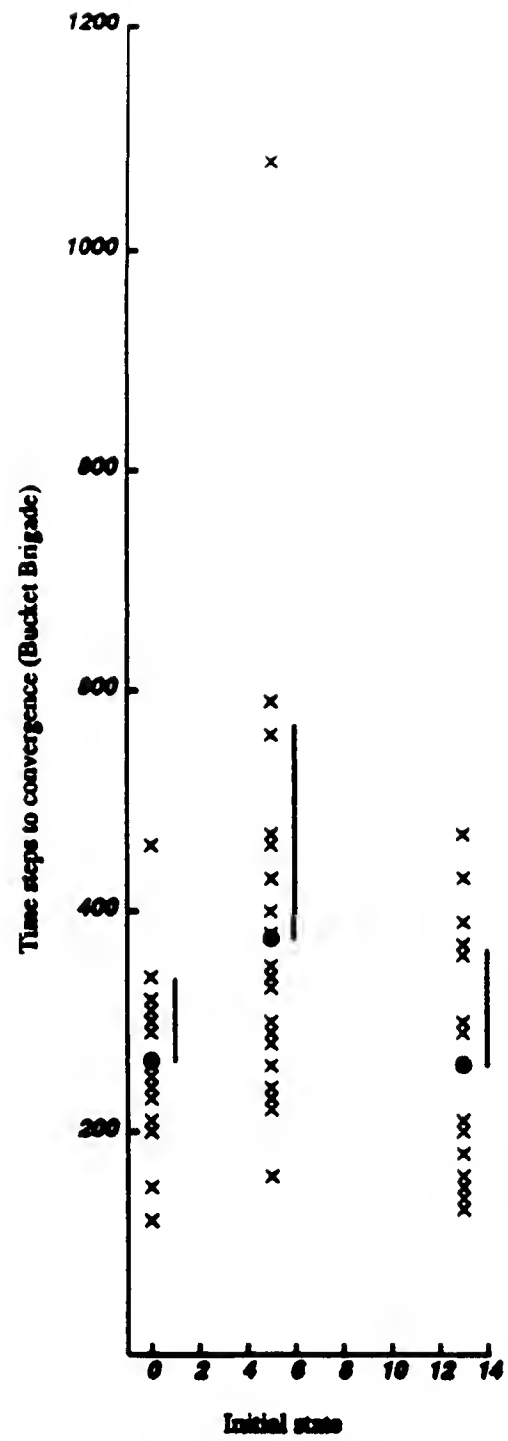

Figure 7: The graph shows the performance of the Bucket Brigade algorithm on three different initial states, one of them the goal state.

learning will be. Sparse reinforcement aggravates the delayed reinforcement problem and increases the number of trials required for the system to discover the correct policy. Pruning the state space by generalizing similar input states is one of the key methods for improving performance of table-driven RL approaches.

Human programmers are excellent at generalization. The reason why it is much easier and faster, even for complex tasks, to hand code a behavior than to learn it, is that learning considers the entire state space of the problem whereas the human designer prunes it very effectively. Usually, the problem of exponential state space is bypassed by a clever ordering of the rules, careful arbitration, and default conditions. None of these options are available in current RL approaches.

[Chapman and Kaelbling 91] and **[Mahadevan and Connell 90]** present complementary approaches to input generalization. The Chapman-Kaelbling approach starts with the most general solution (a single, most general state) and splits it iteratively, based on statistics accumulated over time. When a bit in a state vector is found to be relevant, the state space is split into two subspaces, one with that bit on, and the other with it off. In contrast, the Mahadevan-Connell method starts with a fully differentiated, specific set of all states, and consolidates them based on similarity statistics accumulated over time. Both processes produce state space trees which are sufficiently differentiated but smaller than the fully exponential space.

The default hierarchies of the CS paradigm are also a means of input generalization. Each instance of the # symbol allows for clustering two states into one, with the flexible grouping potential of full generality (all #'s) to full specificity (all non-#'s). Default hierarchies organize the specific rule instances in a system, and speed up the learning process, as previously described. If a CS starts with a single completely general classifier, it will generate more specific rules over time. These rules will be grouped into a default hierarchy based on the relevance of individual bits which are changed from #'s to specific values. This process is neatly analogous to the [Chap-

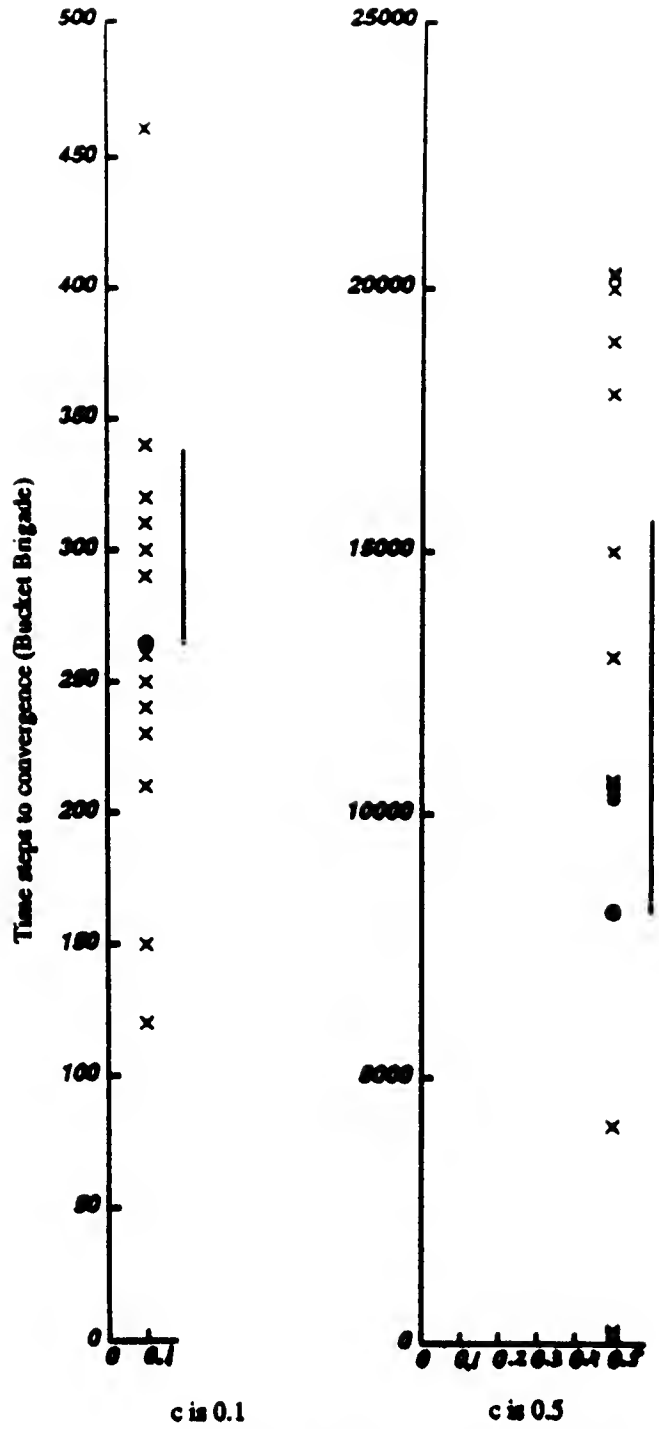

Figure 8: The performance of Bucket Brigade with two different values of c, the bid scaling parameter. The larger value of $c = 0.5$ slows the learning down by more than an order of magnitude.

man and Kaelbling 91] method of bit splitting which also begins with the most general state and subdivides and clusters the more specific states. The CS approach is a more powerful extension, since it eliminates the need for individual bit relevance. The process of generating more specific classifiers implements a power set of the state space. However, it does so incrementally thus never needing to keep around a large number of classifiers, as would a power set implementation of any bit-relevance algorithm akin to [**Chapman and Kaelbling 91**].

The genetic component of the classifier system, which generates new rules and eliminates weak ones, implicitly implements the statistics that are kept explicitly in [**Chapman and Kaelbling 91**]. While the latter must apply some statistical analysis of the gathered data, CS simply works by trial and error until the proper population of appropriate specificity classifiers is evolved. It would be interesting to empirically compare the performance of the two methods on a common problem.

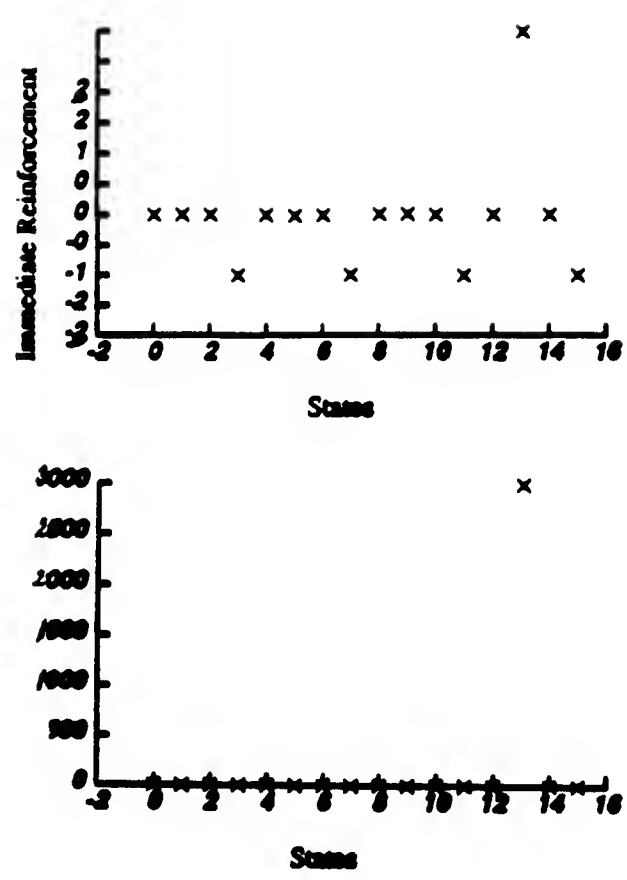

Figure 9: The top graph shows the 3-valued reinforcement function used for testing the two learning algorithms. The bottom graph shows the impulse function which worked well with the standard Bucket Brigade but significantly slowed down both Q-learning and SBB.

The input generalization problem is also addressed by the connectionist literature [**Hinton 90**]. A key difference between connectionist approaches and the RL paradigm is that while RL schemes can be entirely semantics-free and statistical, the generalization criteria are hand coded and therefore understood. In the connectionist case, the representations generated by the networks are not meaningful or usable for the designer. Furthermore, they cannot be debugged if the generalization process fails; and the only solution is to tune the various parameters until the right generalization is found. The key difference, then, is that the generalisation in RL (both in Q-learning and CS) is built in, whereas in connectionist approaches it is a result of the network dynamics.

Paradoxically, it is precisely the unwieldy, fully-exponential quality of the RL state spaces that gives them one of their main positive properties: asymptotic completeness. While hand coded reactive policies take advantage of the cleverness of the designer, they are almost never provably complete. Most irrelevant input states are easily eliminated, but potentially useful ones can be overlooked. Complete state spaces, on the other hand, guarantee that the agent will, given sufficient time and sufficiently rich reinforcement, produce a provably complete policy. However, this quality is of little use if the world is dynamic or the state space is large.

6.1 Modularization

The problem of learning the optimal policy can be cast as searching for paths in the action space which con-

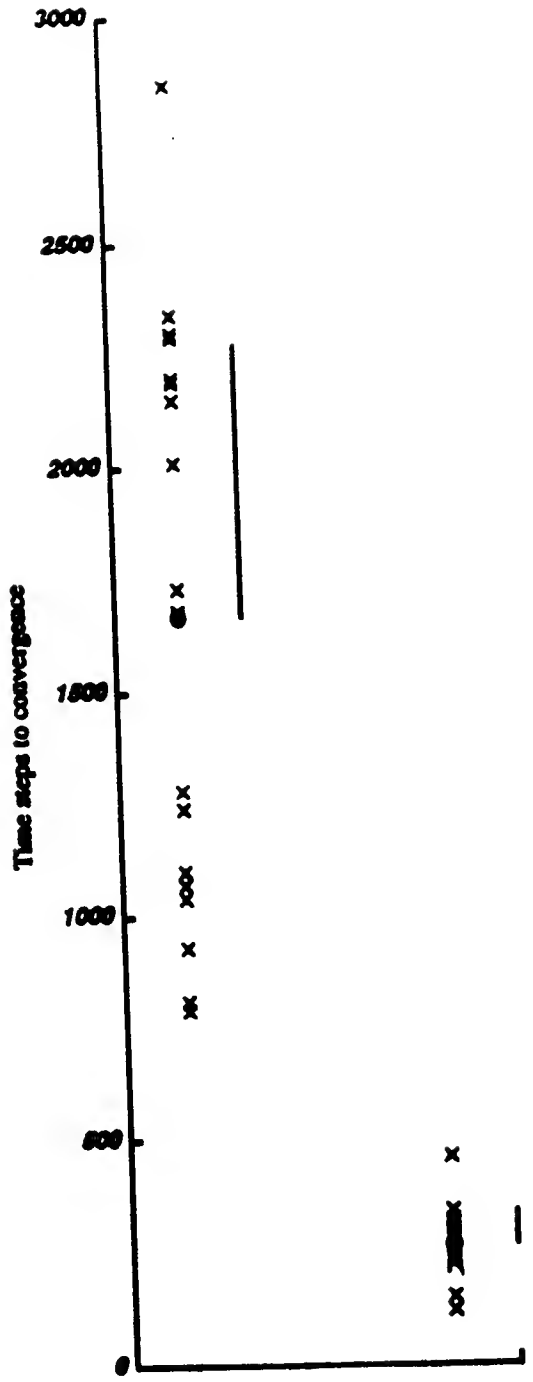

Figure 10: A plot comparing the performance of Q-learning (on the left) and the SBB algorithm (on the right) on the same learning problem.

nect the current state with the goal. The longer the distance between a state and the goal, the longer it takes to learn the policy or the path. This is why policies for large state spaces in Q-learning, and long classifier sequences in the Bucket Brigade, both take a long time to be learned. Breaking the problem into modules or subproblems effectively shortens the distance between the reinforcement signal and the individual actions. Consequently, the length of action sequences to be learned is decreased. However, breaking the problem up into an appropriate set of modules requires domain information about the particular learning task.

[**Mahadevan and Connell 90**] give an example of breaking up a box pushing task into three modules, effectively introducing three subgoals into the learning task. The three are carefully chosen to be orthogonal and non-conflicting, based on the particular task. The robot's behavior repertoire is designed so that whatever state it is in, it is pursuing one of the subgoals: finding a box, pushing a box, or getting unstuck. The reinforcement depends on which of the subgoals is being pursued, but it is available more frequently, since the distance between any state and one of the subgoals, is decreased. Not surprisingly, when tested in both simulation and on the real robot, the modular approach far outperforms the monolithic design in which the robot is only rewarded for actually maintaining contact with and pushing a box.

It is unlikely that any universal strategy for dividing the task into modules exists. However, it would be useful to derive a few principles for task decomposition for particular classes of learning problems. Another interesting question is whether the modularization of a task is dependent on the learning algorithm, i.e. whether there exists some "optimal" set of modules which is independent of the way the modules are learned, but is tied instead to the semantic definition of the problem.

7 Built in Structure and Knowledge

It is often said that "one cannot learn anything unless one almost knows it already" [**Winston 84**]. The tradeoff between the type and amount of built in versus learned information is the key issue in machine learning. The less structure is built in, the more is left to the algorithm to discover. Minimizing built in structure in order to ease the programming task and reduce the learning bias often results in over-specificity and narrowness. It makes the learning process slower, the space and time complexity larger, and the result more task-specific. Neural networks are an example of this type of data-driven learning, biased only by the structure of the network and the training set. These methods have been shown to be sensitive to initial conditions [**Kolen and Pollack 90**], very specific, and of limited ability to generalize [**Hinton 90**].

On the other end of the data-knowledge spectrum lie knowledge-based or knowledge-driven learning schemes. They employ some form of a domain theory in order to minimize the amount of deduction left to the agent, as well as the amount of new information needed from the world. Explanation based learning (EBL) [**Dejong and Mooney 86**] and explanation based generalization (EBG) [**Mitchell et al 86**], [**Mitchell et al 89**] belong in this category. These approaches are constrained by the structure and amount of information provided by the domain theory, and rely on its completeness and accuracy. These properties have earned them the label of "strong" methods as compared with "weak" connectionist approaches [**Hinton 90**].

Reinforcement learning is situated between the two extremes of the spectrum, much closer to the data-driven end. Unlike both the connectionist approaches and EBL-style methods, which require an explicit teacher, RL is unsupervised. Consequently, it is well suited for adaptive agents acting in changing, possibly nondeterministic worlds. Eliminating the teacher removes any bias that might be present in the training set. On the other hand, RL approaches rely on the environment to encode and manifest an observable and learnable mapping between the states the agent can perceive and the actions it can perform. The dependence on the environment rather

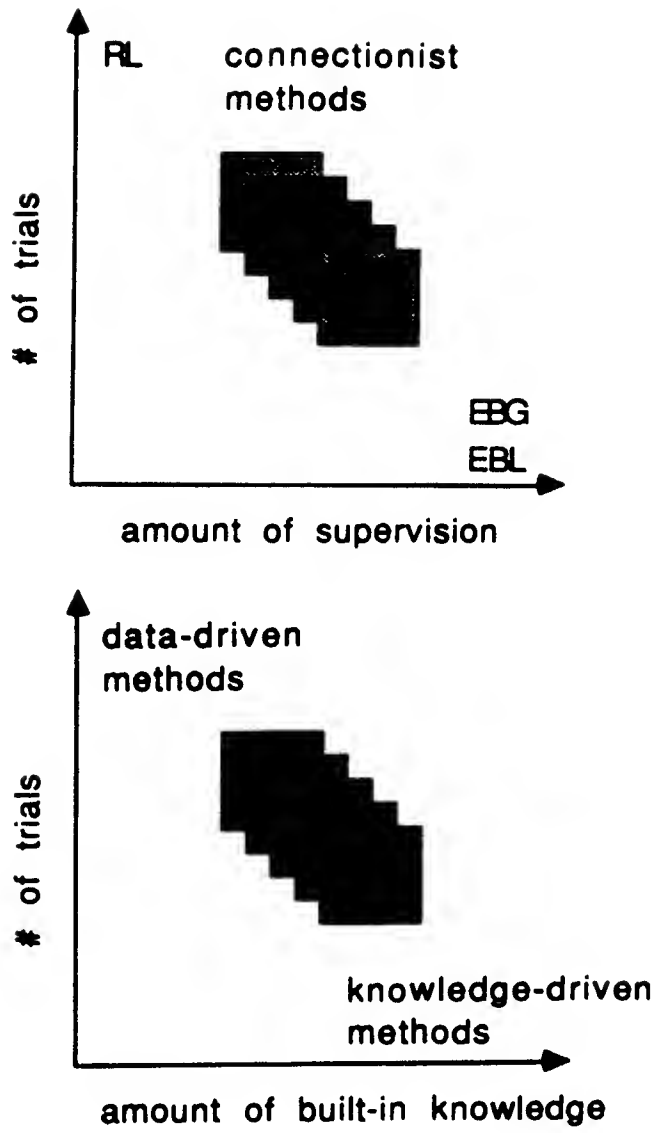

Figure 11: This figure illustrates the relationship between the amount and type of experimental trials in different learning methodologies. The shaded area indicates the desirable properties for a learning algorithm.

than the training set can be recast as the reliance on the designer to properly structure the perceptual apparatus and the reinforcement function.

In order to avoid preprocessing the data, RL approaches manipulate the "raw" input vector. To establish a correlation between each state and the desired action, the algorithms search through the entire space of state-action combinations ($O(2^n|a|)$) requiring a large number of trials to find the optimal policy. In contrast, knowledge-driven learning approaches rely on very few carefully constructed examples, since they encode most of the domain knowledge in the system *a priori*.

Intuitively, the amount of built in structure and the number of training trials are inversely proportional. Additionally, the quality of the gained information affects the required number of additional trials. While knowledge-driven systems require only a few special examples, connectionist systems use a large number of somewhat biased examples, and RL systems depend on many trials which can vary in both accuracy and relevance (figure 11).

8 Summary

This paper analyzed the reinforcement learning problem with respect to two specific algorithms: Q-learning and classifier systems using the Bucket Brigade. The principal weaknesses of RL were discussed: the large time and space complexity, input generalization, and the lack of built in structure.

A main problem with RL approaches is their "unstructured" utilization of the inputs. Since no domain information is used, the entire space of state-action pairs must be explored. Consequently, these algorithms scale poorly with the number of input bits. However, a wealth of sensory information is a key to intelligence, so any future directions in learning must be helped, rather than hurt, by increased amounts of information.

Learning can serve at least two different purposes in a situated agent. It can ease the programmers job by having the agent learn its own behaviors. It can also keep the agent adaptive to a changing world. So far, RL has not fulfilled either of those roles. Learning is a poor substitute for programming any real system because it is overwhelmingly complex and slow. Additionally, not enough is known about the internal dynamics of the parameter interaction, which demands a lot of parameter tuning. It is not yet clear that tuning learning parameters is easier than tuning programming parameters in a hand coded nontrivial agent.

The adaptation property of learning agents is indisputably useful. However, current RL algorithms are very slow to converge to a policy and consequently slow to adapt. Perhaps more importantly, no RL work so far has demonstrated the ability to use previously learned knowledge to speed up the learning of an entirely new policy. Instead, the agents must either start from scratch, or worse, the current policy may be a detriment to learning the next one. Consequently, it has not yet been shown that agents using RL can adapt to more than a single policy.

In spite of its weaknesses, reinforcement learning has been demonstrated to perform well in certain types of tasks and environments. Better understanding those tasks, attempting a large number of versatile experiments of nontrivial agents, and further characterizing the real applications of the approach ought to be be the focus of further RL research.

Acknowledgements

I'm grateful to Lynn Stein, Tom Knight, and Pattie Maes for comments on previous drafts of this paper, and to Rich Sutton and Jose Robles for valuable discussions.

References

[Chapman and Kaelbling 91] "Input Generalization in Delayed Reinforcement Learning: An Algorithm and Performance Comparisons", David Chapman and Leslie P. Kaelbling, *Proceedings, IJCAI-91*, Sydney, Australia, 726-731.

[Dejong and Mooney 86] "Explanation-Based Learning: An Alternative View", Gerald Dejong and Raymond Mooney, *Machine Learning*, 1986, 1:145-176.

[Goldberg 89] "Genetic Algorithms in search, optimization, and machine learning", David E. Goldberg, *Reading, MA, Addison-Wesley*, 1989.

[**Goldberg 85**] "Genetic Algorithms and Rule Learning in Dynamic System Control", David E. Goldberg, *Proceedings of an International Conference on genetic Algorithms and Their Applications*, Pittsburgh, PA, 8-14.

[**Hinton 90**] "Connectionist Learning Procedures", Geoffrey E. Hinton, *in Machine Learning, An Artificial Intelligence Approach*, Vol. 3, Kodratoff and Michalski, eds., 1990, pp. 555-610.

[**Holland 86**] "Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems", John H. Holland, *Machine Learning: An Artificial Intelligence Approach, Vol. 2*, R. S. Michalski, J. G. Carbonell and T. M. Mitchell, eds. Los Altos, CA: Morgan Kaufmann, 1986.

[**Holland 85**] "Properties of the bucket brigade algorithm", John H. Holland, *Proceedings of an International Conference on genetic Algorithms and Their Applications*, Pittsburgh, PA, 1-7.

[**Kaelbling 90**] "Learning in Embedded Systems", Leslie P. Kaelbling, *PhD thesis, Stanford University*, 1990.

[**Kolen and Pollack 90**] "Back Propagation is Sensitive to Initial Conditions", John F. Kolen and Jordan B. Pollack, *Ohio State University AI Lab TR 90-JK-BPSIC*, 1990.

[**Mahadevan and Connell 91**] "Automatic Programming of Behavior-based Robots using Reinforcement Learning", Sridhar Mahadevan and Jonathan Connell, *Proceedings, AAAI-91*, July 1991.

[**Mahadevan and Connell 90**] "Automatic Programming of Behavior-based Robots using Reinforcement Learning", Sridhar Mahadevan and Jonathan Connell, *IBM T. J. Watson Research Center Research Report*, December 5, 1990.

[**Mitchell et al 89**] "Toward a Learning Robot", Tom M. Mitchell, Matthew T. Mason, and Alan D. Christiansen, *CMU-CS-89-106*, 24 January 1989.

[**Mitchell et al 86**] "Explanation-Based Generalization: A Unifying View", Tom M. Mitchell, Richard M. Keller and Smadar T. Kedar-Cabelli, *Machine Learning*, 1986, 1:47-80.

[**Riolo 87**] "Bucket Brigade Performance: I. Long Sequences of Classifiers", Rich L. Riolo, *Proceedings of an International Conference on genetic Algorithms and Their Applications*, Pittsburgh, PA, 184-195.

[**Samuel 59**] "Some studies in machine learning using the game of checkers", A. L. Samuel, *IBM Journal of Research and Development*, Vol. 3, 1959, 211-229.

[**Sutton 88**] "Learning to predict by method of temporal differences", Rich Sutton, *The Journal of Machine Learning*, 3(1), 1988, 9-44.

[**Watkins 89**] "Learning from Delayed Rewards", C. Watkins, *King's College PhD Thesis*, May 1989.

[**Wilson 87**] "Hierarchical credit allocation in a classifier system", Stewart W. Wilson, *Genetic algorithms and simulated annealing*, L. D. Davis, ed. London, Pitman, 1987.

[**Winston 84**] "Artificial Intelligence", Patrick H. Winston, *2nd edition, Addison Wesley*, 1984, pp. 413.